Revisiting Epsilon Serializabilty to improve the Database State Machine (Extended Abstract)*

A. Correia Jr. A. Sousa L. Soares F. Moura R. Oliveira
Universidade do Minho

1 Introduction

Recently, a large body of research has been exploiting group communication based techniques to improve the dependability and performance of synchronously replicated database systems [8, 7, 13, 9]. Database replication based on group communication appears as a promise to overcome the scalability and performance problems of traditional strong consistency protocols, reducing the interactions among the replicas and eliminating deadlocks.

Protocols such as those presented in [8, 7, 13, 9], and in particular the Database State Machine (DBSM), allow a transaction to be executed at any site and postpone the interaction among distributed concurrent transactions, which can be seen as an optimistic execution. Upon receiving the commit request, they propagate relevant information of the transaction to all replicas. If conflicts arise among concurrent transactions, the order in which the transactions were delivered is used to decide which of them commit or abort. The transaction propagation relies on an atomic multicast primitive [6] which guarantees that the sequence of transactions is the same at all non-faulty replicas.

Unfortunately, the optimistic execution of transactions combined with the strictness of the serializability consistency criterion [2] adopted in the DBSM may lead to a considerable number of aborts. In this paper, we investigate how to relax the consistency criteria of DBSM in a controlled manner according to the *Epsilon Serializability* (ESR) concepts [16] and evaluate the direct benefits in terms of performance.

Basically, ESR relies on the assumption that some transactions may tolerate a certain degree of imprecision to improve the overall performance. It allows controlled inconsistencies using a framework that can be in part applied regardless of the application semantics. For instance, a transaction that retrieves a warehouse's amount of sales may accept a value that does not represent the amount in the last millisecond but some value in the last couple of seconds.

* Research funded by FCT, STRONGREP project (FCT POSI / CHS / 41285 / 2001).

To evaluate the benefits of our approach, we use a workload based on the TPC-C [15] benchmark annotating the transactions with the allowed degree of inconsistency.

The rest of this paper is organized as follows. In Section 2, we introduce the concepts behind ESR. In Section 3, we introduce the DBSM and show how it can be augmented in order to incorporate the ESR concepts. In Section 4, we present a set of experiments conducted in order to evaluate our approach. In Section 5, we conclude the paper.

2 Epsilon Serializability

2.1 Motivation

Serializability (SR) [2] defines the standard notion of correctness in transaction processing. Roughly speaking, the execution of concurrent transactions is serializable if it produces the same output and has the same effect on the database as some serial execution of the same transactions. In replicated databases, the databases should behave like a single database, the standard notion of correctness is called one-copy serializability (1SR), which means that the interleaved execution of the transactions must be equivalent to a serial execution of those transactions on a single database.

Unfortunately, this criterion may be sometimes too restrictive when considering a high number of concurrent transactions. It may lead to reduce the overall performance and the effective number of concurrent transactions [5]. Databases usually provide the serializability model as a possible isolation level [1] along with others which can be used to relax the consistency criterion and as a way to improve performance. For instance, one could use a "read uncommitted" isolation level, allowing the transaction to read uncommitted data and produce unbounded inconsistent results. Specifically, in the case of read-only transactions, some databases provide a "snapshot" isolation level or "multi-version". [1] Generally speaking, the database guarantees that the transaction reads data from a snapshot of the

[1] The "Snapshot" isolation level is actually a "multi-version" concurrency control algorithm [4].

	ImpLimit	ExpLimit
T	= 0	= 0
Q^{ϵ}	≥ 0	= 0
U^{ϵ}	= 0	≥ 0

Table 1. Epsilon Transactions

	Q^{ϵ}	U_r^{ϵ}	U_w^{ϵ}
Q^{ϵ}	AOK	AOK	LOK-1
U_r^{ϵ}	AOK	AOK	-
U_w^{ϵ}	LOK-2	-	-

Table 2. Lock Compatibility

committed data at the time it starts, which means that committed data produced during the execution is not visible. In contrast with the choice of the "read uncommitted" isolation level and other levels, the snapshot isolation level presents a stronger criterion. Unluckily, the snapshot isolation level or multi-version concurrency control policies are not available in most databases [2, 4].

2.2 Protocol Overview

In order to alleviate or even overcome these limitations, ESR relies on the assumption that some transactions may tolerate a certain degree of imprecision to improve performance and, as a consequence, reduce the number of aborts. It allows limited and controlled inconsistencies using a framework that can in part be applied regardless of the application semantics. For instance, a transaction that retrieves the amount of sales of a warehouse may accept a value that does not represent the amount in the last millisecond but some value in the last couple of seconds.

ESR suggests the following steps: (*i*) to annotate the transactions using a declarative language to state the inconsistencies allowed; (*ii*) to enhance the concurrency control mechanism (CC) to guarantee that inconsistencies are below the annotated limit for each transaction and (*iii*) to take the appropriate actions if the limit was reached. In what follows, we consider a CC mechanism based on the strict two-phase locking [2].

Transactions are annotated as depicted in Table 1. For read-only epsilon transactions, denoted Q^{ϵ}, we declare the inconsistency that each transaction can *import*. This is expressed as the *ImpLimit* variable. For update epsilon transactions, U^{ϵ}, we declare the inconsistency that the transaction can *export*. That is, the degree of interference it can inflict on read-only transactions. This is expressed as the *ExpLimit* variable.

Similarly to a CC mechanism, a divergence control mechanism (DC) is used to ensure an ESR execution. It can be built systematically augmenting the CC mechanism. Table 2 presents the lock acquisition behavior for the ESR model. At the left (in white) are the locks we want to acquire and at the top (in grey) the locks that have been granted. An epsilon update transaction U^{ϵ} is divided in two components: a read (U_r^{ϵ}) and a write (U_w^{ϵ}). The expression AOK is used when the locks are always compatible, which means that different transactions that attempt to acquire these locks on the same data item always have their requests granted. Dashes represent incompatible locks, which means that different transactions cannot have these locks at the same time on the same data item. LOK means that the locks can be granted depending on the limits of inconsistencies.

In the SR model, when a data item has a write lock and a request from another transaction to read the same item arrives, the request blocks because the locks are incompatible. In the ESR model, the lock may be granted ($LOK-1$), which means that it allows the transaction to read uncommitted data. In the SR model when a data item has one or more read locks and a request from another transaction to update the same item arrives, the request blocks because the locks are incompatible. In the ESR model, the lock may be granted ($LOK-2$), which means that updates can overwrite data that is being read by the queries.

The DC mechanism must act as follows to evaluate if it is possible to grant the non-SR locks. First, it must define for each Q^{ϵ} a variable that accumulates the amount of inconsistency imported, e.g. *acum.ImpLimit*, and for each U^{ϵ} a variable that accumulates the amount of inconsistency exported, e.g. *acum.ExpLimit*. In the case of $LOK-1$, the DC mechanism must check if the inconsistency does not force Q^{ϵ} to exceed its limit, i.e. $ImpLimit \leq acum.ImpLimit$, or U^{ϵ} to exceed its limit, i.e. $ExpLimit \leq acum.ExpLimit$. If both limits are not exceeded the lock is granted, otherwise it is not. In the case of $LOK-2$, the DC mechanism must also take into account that the read locks can be granted for more than one Q^{ϵ} and hence the DC mechanism must compute the limits for each Q^{ϵ}.

Each transaction may have different variables to define limits, expressing the number of non-SR conflicts, absolute values (e.g., the tolerated imprecision in the amount of products in stock) and the age of the information (i.e., the time interval that a epsilon-transaction can process using stale information). If a limit was defined based on the number of non-SR conflicts, the DC mechanism would simply increment a counter to express the conflict. If it was defined based on absolute values, the increment of the accumulators would take into account the absolute differences between the data item before and after the update. In the case of the age of the information, Q^{ϵ} does not exceed the specified limit if $ImpLimit + firstTime \leq now()$, where $firstTime$ represents the first time that the transaction read uncommitted data and $now()$ represents current time. This test must be applied upon requesting commit, which means that the age implies the time interval that Q^ϵ could execute using stale data. If the expression evaluates to false, Q^ϵ must abort. For the age information, U^ϵ's $ExpLimit$ can be set to ∞, since the amount of inconsistency exported is not considered in this case because the test is applied just upon requesting commit.

2.3 Update Transactions

In order to further improve performance and further reduce the number of aborts, it would be interesting to allow update transactions to import inconsistency. However, in this case, the DC mechanism itself is not sufficient to guarantee a consistent state of the database since it can only limit inconsistencies in the scope of a single epsilon-transaction. Successive epsilon-transactions may compound the inconsistency on particular data items and therefore introduce an arbitrarily large divergence. In [12], it studies the divergence problem and proposes two types of consistency restoration techniques: compensation-based and independent updates. For the sake of completeness, we briefly present the intuition behind the restoration techniques. However, our approach does not currently encompass update transactions to import inconsistency. Further details can be found in [12, 3, 11, 10].

Compensation-based techniques attempt to restore state consistency by undoing transactions and, afterwards, rescheduling these transactions in such a manner that it minimizes the number of aborts. This approach is undesirable to our goals since the assumption of being able to redo or compensate external effects is often impractical. Furthermore, this technique usually requires knowledge about the application semantics to define the compensation activities and to reschedule the transactions. In [11], it is presented an approach that automatically extracts information from transactions in order to correctly reschedule them and therefore reduce the number of aborts.

The independent updates technique borrows heavily from reservation techniques. Reservation techniques basically attempt to avoid conflicts [10].

3 Epsilon serializability and the DBSM

The Database State Machine [8] is based on the deferred update replication technique [2] which reduces the need for distributed coordination among concurrent transactions during their execution. A transaction is locally synchronized at the database where it initiated according to some CC mechanism [2]. From a global point of view, the transaction execution is optimistic since there is no coordination with any other database site possibly executing some concurrent transaction. Interaction with other database sites on behalf of the transaction only occurs when the commit is requested, i.e. when t enters the *committing state*. At this point, a termination protocol is started: $i)$ the transaction's relevant information is atomically propagated to all database sites, and $ii)$ each database site certifies the transactions determining its fate: commit or abort.

In order for a database site to certify a committing transaction t the site must be able to determine which transactions conflict with t. A transaction t' *conflicts with* t if: $i)$ t and t' have conflicting operations and $ii)$ t' does not *precede* t.

Two operations conflict when they are issued by different transactions, access the same data item and at least one of them is a write operation.

The precedence relation between transactions t and t' is denoted $t' \rightarrow t$ (i.e., t' precedes t) and defined as: $i)$ if t and t' execute at the same database site, t' precedes t if t' enters the committing state before t; or $ii)$ if t and t' execute at different sites, for example s_i and s_j, respectively, t' precedes t if t' commits at s_i before t enters the committing state at s_i.

From a global point of view, the DBSM uses the certification procedure as a CC mechanism. However, a read-only transaction (Q^ϵ) is entirely handled locally and does not interfere with remote transactions, which means that remote transactions are not certified against local transactions. In contrast with that, a remote transaction that was committed must locally store information, which is done atomically acquiring the necessary locks and afterwards updating it. Certainly, the remote transactions interfere with local running transactions and usually as a consequence the local transactions are aborted. To augment the DBSM with the ESR model for Q^ϵ transactions, we need to only adjust the local CC mechanism accordingly to Section 3.

We can proceed as follows:[2] During lock acquisition, the local DC mechanism verifies if the inconsistency introduced by any transaction committed since Q^ϵ began does not force Q^ϵ to exceed its limits, i.e. $ImpLimit \leq acum.ImpLimit$, or the remote U^ϵ to exceed its limits, i.e. $ExpLimit \leq acum.ExpLimit$. If both limits are not exceeded the local Q^ϵ can continue and the U^ϵ can update the database. Otherwise, it aborts the local Q^ϵ.

4 Experiments

To evaluate our approach, we used a simulation tool that combines real and simulated code allowing us to test prototypes early and with great flexibility. Specifically, the database, the clients and the network are simulated. The

[2]Besides the solution based on the ESR, it would be possible to wait until the conclusion of the Q^ϵ. However, in this case, we could introduce an arbitrary latency in the system which might not be desirable.

group communication and the DBSM protocols are real implementations.

The simulated system consists of 9 database sites which communicate through a LAN with an average bandwidth of 1 Gbps and 1 ms of latency. Each site has a single processor comparable with a Pentium III at 1 GHz and a storage with 9.5 MBps of throughput for blocks of 4 KB. Each site has a full replica of the database. Clients are uniformly distributed across 3 of the database sites.

The application profile used is based on TPC-C [15], the industry standard on-line transaction processing benchmark. TPC-C mimics a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. It has the following relations: (*i*) *warehouse*, (*ii*) *district*, (*iii*) *customer*, (*iv*) *stock*, (*v*) *orders*, (*vi*) *order line*, (*vii*) *history*, (*viii*) *new order* and (*ix*) *item*.

The traffic is a mixture of read-only and update intensive transactions. A client can request five different transactions types as follows: New Order, adding a new order into the system (with 44% probability of occurrence); Payment, updating the customer's balance, district and warehouse statistics (44%); Order Status, returning a given customer latest order (4%); Delivery, recording the delivery of products (4%); Stock Level, determining the number of recently sold items that have a stock level below a specified threshold (4%).

Notice that, according to TPC-C, an additional warehouse should be configured for each additional 10 clients. The initial size of tables depends on the number of configured clients. Notice also that the TPC-C is being used only as the basis for a realistic application scenario in order to evaluate our approach and not as a benchmark. The constraints required for throughput, performance, response time, screen load and background execution of transactions are not considered here and thus the results are not comparable with other system results obtained with TPC-C.

In what follows, we show how the ESR can improve the performance of the TPC-C transactions and also reduce the abort rate. Transactions are annotated with the amount of conflicts tolerated. For a detailed description and evaluation of the simulation tool the reader is referred to [14].

Figure 1 depicts the results obtained. We ran a set of simulations with strict two-phase locking for the serializable model (SR), with an extended version of the strict two-phase locking for the ESR model (ESR), and with multi-version concurrency control (MV). For each set of experiments we varied the number of clients from 250 to 1000 and measured the abort rate (Figure 1(a)), the latency (Figure 1(b)) and the throughput of the system (Figure 1(c)). For the ESR runs in Figures 1(a), 1(b) and 1(c), we assumed no limits on the amount of allowed inconsistency that could be imported.

Figure 1(a) shows the abort rate exhibited by the four set of runs. In the DBSM, aborts may result from conflicts between local transactions or from conflicts arising from the certification of remote transactions. Due to the strong locality of the TPC-C workload and its way of distributing the clients among the sites, operations that access information from remote warehouses represent less than 10% and, therefore, aborts of transactions due to certification tend to be negligible. As such, the aborts constituting the graph of Figure 1(a) correspond to local transaction conflicts (i.e., write-write conflicts) and, as expected, can only arise when using a multi-version CC mechanism. With both strict two-phase locking CC or the DC mechanism, write locks need to be acquired and conflicting transactions block instead of *a posteriori* abort.

Figure 1(b) shows the latency as observed by the clients.[3] Up to 750 clients, the latency of the system is kept very low and any differences among the experiments are absorbed by variance. With more than 750 clients, SR presents the highest latency which is explained by the contention introduced with locking. The advantage of using epsilon serializability directly translates on latency. In the ESR runs, for 1000 clients, a gain of around 40% is obtained with respect to SR. This, however, shall be regarded as a lower bound on latency for epsilon serializability applied to read-only transactions as we are admitting unbounded inconsistency here.

Figure 1(c) shows a indistinguishable performance of the considered CC mechanisms regarding throughput. It is noticeable a reduction on performance above 750 clients for all set of experiments. However, the think-time and keying-time constraints imposed by the TPC-C [15] conceal any performance differences that could be yielded by differences on the latency of the different CC approaches.

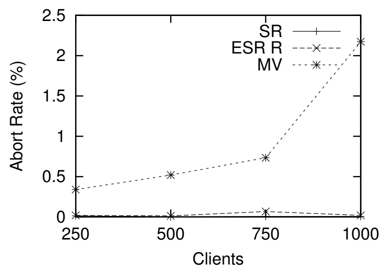
Finally, in Figure 1(d) we vary the number of allowed conflicts. Until 750 clients, the values are quite similarly since there is no contention problems. Above this value, the ESR - E999 (i.e., Q^{ϵ} and U^{ϵ} tolerate 999 non-SR conflicts) presents values similar to unlimited ϵ (Figure 1(b)). Reducing the number of allowed conflicts increases latency and the ESR - E001 (i.e., Q^{ϵ} and U^{ϵ} tolerate 1 non-SR conflicts) is identical to SR.

5 Conclusion

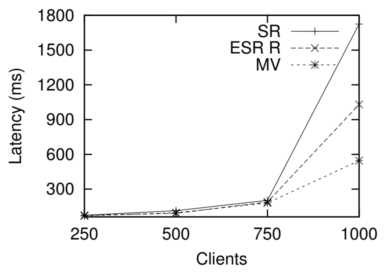
This paper presents early results on the use of Epsilon Serializability (ESR) [16] to *stretch* the consistency criteria of the DBSM in order to improve its performance. Basically, ESR relies on the assumption that some transactions may tolerate a certain degree of imprecision to improve performance or to reduce the number of aborts.

We have conducted a set of experiments using TPC-C [15] as workload. Our results show that indeed ESR can

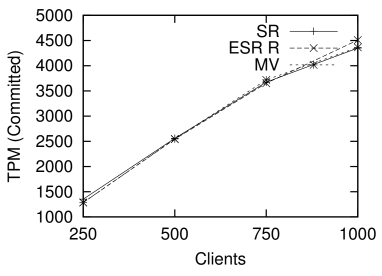
[3]For both latency and throughput measurements, only committed transactions were taken into account.



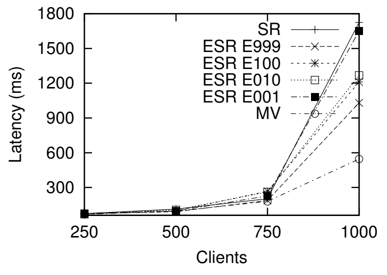
(a) Abort rate



(b) Latency



(c) Throughput



(d) Latency - Conflicts

Figure 1. Performance

be used in a controlled and consentaneous way, establishing a trade-off between consistency and performance. Varying the number of conflicts, allowing the read of stale data, we managed to reduce transaction latency up to 40%. As expected, for the epsilon transactions, latency is up bounded by strict locking policies and bottom bounded by multi-version concurrency control. In contrast with multi-version, epsilon serializability eliminates the aborts generated by write-write conflicts.

The next step on this work will be the support of reservation techniques to allow to stretch the consistency of update transactions.

References

[1] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *ACM SIGMOD International Conference on Management of Data*, 1995.

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] P. Drew and C. Pu. Asynchronous consistency restoration under epsilon serializability. Technical report, Department of Computer Science, Hong Kong, University of Science and Technology., 1993.

[4] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. Technical report, NSF Grant IRI 97-11374, 2004.

[5] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *ACM SIGMOD International Conference on Management of Data*, 1996.

[6] V. Hadzilacos and S. Toueg. A Modular Approach to Fault-Tolerant Broadcasts and Related Problems. Technical report, Cornell University, 1994.

[7] B. Kemme and G. Alonso. A Suite of Database Replication Protocols Based on Group Communication Primitives. In *IEEE International Conference on Distributed Computing Systems*, 1998.

[8] F. Pedone. *The Database State Machine and Group Communication Issues*. PhD thesis, Département d'Informatique, l'École Polytechnique Fédérale de Lausanne, 1999.

[9] R. Jiménez Peris, M. Patiño Martínez, B. Kemme, and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. *IEEE International Conference on Distributed Computing Systems*, 2002.

[10] N. Preguiça, J. Martins, M. Cunha, and H. Domingos. Reservations for Conflict Avoidance in a Mobile Database System. In *Proceedings of The First International Conference on Mobile Systems, Applications, and Services*, 2003.

[11] N. Preguiça, M. Shapiro, and J. Martins. SQLIceCube: Automatic Semantics-base Reconciliation for Mobile Databases. Technical report, Instituto Superior Técnico, Lisboa, Portugal, 2003.

[12] C. Pu. Generalized transaction processing with epsilon-serializability. In *Proceedings of Fourth International Workshop on High Performance Transaction Systems*, 1991.

[13] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial Replication in the Database State Machine. In *IEEE International Symposium on Network Computing and Applications*, 2001.

[14] A. Sousa, J. Pereira, L. Soares, A. Correia Jr., L. Rocha, R. Oliveira, and F. Moura. Testing the Dependability and Performance of GCS-Based Database Replication Protocols. Technical report, Departamento de Informática, Universidade do Minho, 2004.

[15] Transaction Processing Performance Council (TPC). TPC benchmark C Standard Specification Revision 5.0, 2001.

[16] K. Wu, P. Yu, and C. Pu. Divergence control for epsilon-serializability. In *Proceedings of 8th International Conference on Data Engineering*, 1992.